4

Object-Oriented Programming in C#

WHAT'S IN THIS CHAPTER?

- Using inheritance with classes and records
- Working with access modifiers
- Using interfaces
- Working with default interface methods
- Using dependency injection
- Using generics

CODE DOWNLOADS FOR THIS CHAPTER

The source code for this chapter is available on the book page at `www.wiley.com`. Click the Downloads link. The code can also be found at `https://github.com/ProfessionalCSharp/ProfessionalCSharp2021` in the directory `1_CS/ObjectOrientation`.

The code for this chapter is divided into the following major examples:

- VirtualMethods
- AbstractClasses
- InheritanceWithConstructors
- RecordsInheritance
- UsingInterfaces
- DefaultInterfaceMethods
- GenericTypes
- GenericTypesWithConstraints

All the projects have nullable reference types enabled.

OBJECT ORIENTATION

C# is not a pure object-oriented programming language because it offers multiple programming paradigms. However, object orientation is an important concept with C#; it's a core principle of all the libraries offered by .NET.

The three most important concepts of object orientation are *inheritance*, *encapsulation*, and *polymorphism*. Chapter 3, "Classes, Records, Structs, and Tuples," talks about creating individual types to arrange properties, methods, and fields. When members of a type are declared `private`, they cannot be accessed from the outside. They are *encapsulated* within the type. This chapter covers inheritance and polymorphism and extends encapsulation features with inheritance.

The previous chapter explained all the members of a type. This chapter explains how to use inheritance to enhance base types, how to create a hierarchy of classes, and how polymorphism works with C#. It also describes all the C# keywords related to inheritance, shows how to use interfaces as contracts for dependency injection, and covers default interface methods that allow implementations with interfaces.

INHERITANCE WITH CLASSES

If you want to declare that a class derives from another class, use the following syntax:

```
class MyDerivedClass: MyBaseClass
{
  // members
}
```

> **NOTE** *If you do not specify a base class in a class definition, the base class will be* `System.Object`.

Let's get into an example to define a base class `Shape`. Something that's common with shapes—no matter whether they are rectangles or ellipses—is that they have position and size. For position and size, corresponding records are defined that are contained within the `Shape` class. The `Shape` class defines read-only properties `Position` and `Size` that are initialized using auto properties with property initializers (code file `VirtualMethods/Shape.cs`):

```
public class Position
{
  public int X { get; set; }
  public int Y { get; set; }
}

public class Size
{
  public int Width { get; set; }
  public int Height { get; set; }
}

public class Shape
{
  public Position Position { get; } = new Position();
  public Size Size { get; } = new Size();
}
```

> **NOTE** *With the shapes sample, the* `Position` *and* `Size` *objects are contained within an object of the* `Shape` *class. This is the concept of composition. The* `Rectangle` *and* `Ellipse` *classes derive from the base class* `Shape`*. This is inheritance.*

Virtual Methods

By declaring a base class method as `virtual`, you allow the method to be overridden in any derived classes.

The following code snippet shows the `DisplayShape` method that is declared with the `virtual` modifier. This method is invoked by the `Draw` method of the `Shape`. Virtual methods can be `public` or `protected`. The access modifier cannot be changed when overriding this method in a derived class. Because the `Draw` method has a `public` access modifier, this method can be used from the outside when using the `Shape` or when using any class deriving from `Shape`. The `Draw` method cannot be overridden as it doesn't have the virtual modifier applied (code file `VirtualMethods/Shape.cs`):

```
public class Shape
{
  public void Draw() => DisplayShape();

  protected virtual void DisplayShape()
  {
    Console.WriteLine($"Shape with {Position} and {Size}");
  }
}
```

> **NOTE** *All the C# access modifiers are discussed later in this chapter in detail.*

You also may declare a property as `virtual`. For a virtual or overridden property, the syntax is the same as for a nonvirtual property, with the exception of the keyword `virtual`, which is added to the definition:

```
public virtual Size Size { get; set; }
```

For simplicity, the following discussion focuses mainly on methods, but it applies equally well to properties.

Methods that are declared virtual can be overridden in a derived class. To declare a method that overrides a method from a base class, use the `override` keyword (code file `VirtualMethods/ConcreteShapes.cs`):

```
public class Rectangle : Shape
{
  protected override void DisplayShape()
  {
    Console.WriteLine($"Rectangle at position {Position} with size {Size}");
  }
}
```

Virtual functions offer a core feature of OOP: *polymorphism*. With virtual functions, the decision of which method to invoke is delayed during runtime. The compiler creates a *virtual method table* (vtable) that lists the methods that can be invoked during runtime, and it invokes the method based on the type at runtime.

For performance reasons, in C#, functions are not virtual by default. For nonvirtual functions, the vtable is not needed, and the compiler directly addresses the method that's invoked.

The `Size` and `Position` types override the `ToString` method. This method is declared as `virtual` in the base class `Object` (code file `VirtualMethods/ConcreteShapes.cs`):

```
public class Position
{
  public int X { get; set; }
  public int Y { get; set; }

  public override string ToString() => $"X: {X}, Y: {Y}";
}

public class Size
{
  public int Width { get; set; }
  public int Height { get; set; }

  public override string ToString() => $"Width: {Width}, Height: {Height}";
}
```

Before C# 9, there was the rule that, when overriding methods of the base class, the signature (all parameter types and the method name) and the return type must match exactly. If you want different parameters, you need to create a new member that does not override the base member.

With C# 9, there's a small change to this rule: when overriding methods, the return type might differ, but only to return a type that derives from the return type of the base class. One example where this can be used is to create a type-safe `Clone` method. The `Shape` class defines a virtual `Clone` method that returns a `Shape` (code file `VirtualMethods/Shape.cs`):

```
public virtual Shape Clone() => throw new NotImplementedException();
```

The `Rectangle` class overrides this method to return a `Rectangle` type instead of the base class `Shape` by creating a new instance and copying all the values from the existing instance to the newly created one:

```
public override Rectangle Clone()
{
  Rectangle r = new();
  r.Position.X = Position.X;
  r.Position.Y = Position.Y;
  r.Size.Width = Size.Width;
  r.Size.Height = Size.Width;
  return r;
}
```

In the top-level statements of the `Program.cs` file, a rectangle and an ellipse are instantiated, properties are set, and the rectangle is cloned by invoking the virtual `Clone` method. Finally, the `DisplayShapes` method is invoked passing all the different created shapes. The `Draw` method of the `Shape` class is invoked to, in turn, invoke the overridden methods of the derived types. In this code snippet, you also see the `Ellipse` class used; this is similar to the `Rectangle` type, deriving from `Shape` (code file `VirtualMethods/Program.cs`):

```
Rectangle r1 = new();
r1.Position.X = 33;
r1.Position.Y = 22;
r1.Size.Width = 200;
r1.Size.Height = 100;
```

```
Rectangle r2 = r1.Clone();
r2.Position.X = 300;

Ellipse e1 = new();
e1.Position.X = 122;
e1.Position.Y = 200;
e1.Size.Width = 40;
e1.Size.Height = 20;

DisplayShapes(r1, r2, e1);

void DisplayShapes(params Shape[] shapes)
{
  foreach (var shape in shapes)
  {
    shape.Draw();
  }
}
```

Run the program to see the output of the `Draw` method coming from the implementation of the overridden `Rectangle` and `Shape DisplayShape` methods:

```
Rectangle at position X: 33, Y: 22 with size Width: 200, Height: 100
Rectangle at position X: 300, Y: 22 with size Width: 200, Height: 200
Ellipse at position X: 122, Y: 200 with size Width: 40, Height: 20
```

> **NOTE** *Neither member fields nor static methods can be declared as virtual. The concept of virtual members doesn't make sense for anything other than instance function members.*

Hiding Methods

If a method with the same signature is declared in both base and derived classes, but the methods are not declared with the modifiers `virtual` and `override`, respectively, then the derived class version is said to *hide* the base class version.

For hiding methods, you can use the `new` keyword as a modifier with the method declaration. In most cases, you would want to override methods rather than hide them. By hiding them, you risk calling the wrong method for a given class instance. However, as shown in the following example, C# syntax is designed to ensure that the developer is warned at compile time about this potential problem, thus making it safer to hide methods if that is your intention. This also has versioning benefits for developers of class libraries.

Suppose that you have a class called `Shape` in a class library:

```
public class Shape
{
  // various members
}
```

At some point in the future, you write a derived class `Ellipse` that adds some functionality to the `Shape` base class. In particular, you add a method called `MoveBy`, which is not present in the base class:

```
public class Ellipse: Shape
{
  public void MoveBy(int x, int y)
```

```
    {
      Position.X += x;
      Position.Y += y;
    }
  }
```

At some later time, the developer of the base class decides to extend the functionality of the base class and, by coincidence, adds a method that is also called `MoveBy` and that has the same name and signature as yours; however, it probably doesn't do the same thing. This new method might be declared `virtual` or not.

If you recompile the derived class, you get a compiler warning because of a potential method clash. The application is still working, and where you've written code to invoke the `MoveBy` method using the `Ellipse` class, the method you've written is invoked. Hiding a method is the default behavior to avoid breaking changes when adding methods to a base class.

To get rid of the compilation error, you need to add the `new` modifier to the `MoveBy` method. The code the compiler is creating with or without the `new` modifier is the same; you just get rid of the compiler warning and flag this as a new method—a different one from the base class:

```
public class Ellipse: Shape
{
  new public void MoveBy(int x, int y)
  {
    Position.X += x;
    Position.Y += y;
  }
  //...
}
```

Instead of using the `new` keyword, you can also rename the method or override the method of the base class if it is declared virtual and serves the same purpose. However, if other methods already invoke this method, a simple rename can lead to breaking other code.

> **NOTE** *You shouldn't use the* `new` *method modifier to hide members of the base class deliberately. The main purpose of this modifier is to deal with version conflicts and react to changes on base classes after the derived class was done.*

Calling Base Versions of Methods

If a derived class overrides or hides a method in its base class, then it can invoke the base class version of the method by using the `base` keyword. For example, in the base class `Shape`, the virtual `Move` method is declared to change the actual position and write some information to the console. This method should be called from the derived class `Rectangle` to use the implementation from the base class (code file `VirtualMethods/Shape.cs`):

```
public class Shape
{
  public virtual void Move(Position newPosition)
  {
    Position.X = newPosition.X;
    Position.Y = newPosition.Y;
    Console.WriteLine($"moves to {Position}");
  }
  //...
}
```

The `Move` method is overridden in the `Rectangle` class to add the term `Rectangle` to the console. After this text is written, the method of the base class is invoked using the `base` keyword (code file `VirtualMethods/ConcreteShapes.cs`):

```
public class Rectangle: Shape
{
  public override void Move(Position newPosition)
  {
    Console.Write("Rectangle ");
    base.Move(newPosition);
  }
  //...
}
```

Now move the rectangle to a new position (code file `VirtualMethods/Program.cs`):

```
r1.Move(new Position { X = 120, Y = 40 });
```

Run the application to see output that is a result of the `Move` method in the `Rectangle` and the `Shape` classes:

```
Rectangle moves to X: 120, Y: 40
```

> **NOTE** *Using the* `base` *keyword, you can invoke any method of the base class—not just the method that is overridden.*

Abstract Classes and Methods

C# allows both classes and methods to be declared as abstract. An abstract class cannot be instantiated, whereas an *abstract* method does not have an implementation and must be overridden in any nonabstract derived class. Obviously, an abstract method is *automatically* virtual. If any class contains any abstract methods, that class is also abstract and must be declared as such.

Let's change the `Shape` class to be `abstract`. Instead of throwing a `NotImplementedException`, the `Clone` method is now declared abstract, and thus it can't have any implementation in the `Shape` class (code file `AbstractClasses/Shape.cs`):

```
public abstract class Shape
{
  public abstract Shape Clone(); // abstract method
}
```

When deriving a type from the abstract base class that itself is not abstract, it's a concrete type. With a concreate class it is necessary to implement all abstract members. Otherwise, the compiler complains (code file `AbstractClasses/ConcreteShapes.cs`):

```
public class Rectangle : Shape
{
  //...
  public override Rectangle Clone()
  {
    Rectangle r = new();
    r.Position.X = Position.X;
    r.Position.Y = Position.Y;
    r.Size.Width = Size.Width;
    r.Size.Height = Size.Width;
    return r;
  }
}
```

Using the abstract `Shape` class and the derived `Ellipse` class, you can declare a variable of a `Shape`. You cannot instantiate it, but you can instantiate an `Ellipse` and assign it to the `Shape` variable (code file `AbstractClasses/Program.cs`):

```
Shape s1 = new Ellipse();
s1.Draw();
```

Sealed Classes and Methods

If you don't want to allow other classes to derive from your class, your class should be sealed. Adding the `sealed` modifier to a class doesn't allow you to create a subclass of it. Sealing a method means it's not possible to override this method.

```
sealed class FinalClass
{
  //...
}

class DerivedClass: FinalClass // wrong. Cannot derive from sealed class.
{
  //...
}
```

The most likely situation in which you'll mark a class or method as `sealed` is if the class or method is internal to the operation of the library, class, or other classes that you are writing. Overriding methods could lead to instability of the code. When you seal the class, you make sure that overriding is not possible.

There's another reason to seal classes. With a sealed class, the compiler knows that derived classes are not possible, and thus the virtual table used for virtual methods can be reduced or eliminated, which can increase performance. The `string` class is sealed. I haven't seen a single application that doesn't use strings, so it's best to have this type as performant as possible. Making the class sealed is a good hint for the compiler.

Declaring a method as `sealed` serves a purpose similar to that for a class. The method can be an overridden method from a base class, but in the following example, the compiler knows another class cannot extend the virtual table for this method; it ends here.

```
class MyClass: MyBaseClass
{
  public sealed override void FinalMethod()
  {
    // implementation
  }
}

class DerivedClass: MyClass
{
  public override void FinalMethod() // wrong. Will give compilation error
  {
  }
}
```

To use the `sealed` keyword on a method or property, the member must have first been overridden from a base class. If you do not want a method or property in a base class overridden, then don't mark it as virtual.

Constructors of Derived Classes

Chapter 3 discusses how constructors can be applied to individual classes. An interesting question arises as to what happens when you start defining your own constructors for classes that are part of a hierarchy, inherited from other classes that may also have custom constructors.

In the sample application that uses shapes, so far, custom constructors have not been specified. The compiler creates a default constructor automatically to initialize all members to `null` or `0` (depending on whether the types are reference or value types) or uses the code from specified property initializers to add these to the default constructor. Now, let's change the implementation to create immutable types and define custom constructors to initialize their values. The `Position`, `Size`, and `Shape` classes are changed to specify read-only properties, and the constructors are changed to initialize the properties. The `Shape` class is still abstract, which doesn't allow creating instances of this type (code file `InheritanceWithConstructors/Shape.cs`):

```
public class Position
{
  public Position(int x, int y) => (X, Y) = (x, y);

  public int X { get; }
  public int Y { get; }

  public override string ToString() => $"X: {X}, Y: {Y}";
}

public class Size
{
  public Size(int width, int height) => (Width, Height) = (width, height);

  public int Width { get; }
  public int Height { get; }

  public override string ToString() => $"Width: {Width}, Height: {Height}";
}

public abstract class Shape
{
  public Shape(int x, int y, int width, int height)
  {
    Position = new Position(x, y);
    Size = new Size(width, height);
  }

  public Position Position { get; }
  public virtual Size Size { get; }

  public void Draw() => DisplayShape();

  protected virtual void DisplayShape()
  {
    Console.WriteLine($"Shape with {Position} and {Size}");
  }

  public abstract Shape Clone();
}
```

Now the `Rectangle` and `Ellipse` types need to be changed as well. Because the `Shape` class doesn't have a parameterless constructor, the compiler complains because it cannot automatically invoke the constructor of the base class. A custom constructor is required here as well.

With the new implementation of the `Ellipse` class, a constructor is defined to supply the position and size for the shape. To invoke the constructor from the base class, such as invoking methods of the base class, you use the `base` keyword, but you just can't use the `base` keyword in the block of the constructor body. Instead, you need to use the `base` keyword in the constructor initializer and pass the required arguments. The `Clone` method can now be simplified to invoke the constructor to create a new `Ellipse` object by forwarding the values from the existing object (code file `InheritanceWithConstructors/ConcreteShapes.cs`):

```
public class Ellipse : Shape
{
  public Ellipse(int x, int y, int width, int height)
    : base(x, y, width, height) { }

  protected override void DisplayShape()
  {
    Console.WriteLine($"Ellipse at position {Position} with size {Size}");
  }

  public override Ellipse Clone() =>
    new(Position.X, Position.Y, Size.Width, Size.Height);
}
```

> **NOTE** *Chapter 3 covers constructor initializers with the* `this` *keyword to invoke other constructors of the same class. To invoke constructors of the base class, you use the* `base` *keyword.*

MODIFIERS

You have already encountered quite a number of so-called modifiers—keywords that can be applied to a type or a member. Modifiers can indicate the visibility of a method, such as `public` or `private`, or the nature of an item, such as whether a method is `virtual` or `abstract`. C# has a number of modifiers, and at this point it's worth taking a minute to provide the complete list.

Access Modifiers

Access modifiers indicate which other code items can access an item.

You can use all the access modifiers with members of a type. The `public` and `internal` access modifiers can also be applied to the type itself. With nested types (types that are specified within types), you can apply all access modifiers. In regard to access modifiers, nested types are members of the outer type, such as those shown in the following code snippet where the `OuterType` is declared with the `public` access modifier, and the type `InnerType` has the `protected` access modifier applied. With the `protected` access modifier, the `InnerType` can be accessed from the members of the `OuterType`, and all types that derive from the `OuterType`:

```
public class OuterType
{
  protected class InnerType
  {
    // members of the inner type
  }
  // more members of the outer type
}
```

The `public` access modifier is the most open one; everyone has access to a class or a member that has the `public` access modifier applied. The `private` access modifier is the most restrictive one. Members with this access modifier can be used only within the class where the modifier is used. The `protected` access modifier is in between these access restrictions. In addition to the `private` access modifier, it allows access to all types that derive from the type where the `protected` access modifier is used.

The `internal` access modifier is different. This access modifier has the scope of the assembly. All the types defined within the same assembly have access to members and types where the internal access modifier is used.

If you do not supply an access modifier with a type, by default `internal` access is specified. You can use this type only within the same assembly.

The `protected internal` access modifier is a combination of `protected` and `internal`—combining these access modifiers with OR. `protected internal` members can be used from any type in the same assembly or from types from another assembly if an inheritance relationship is used. With the intermediate language (IL) code, this is known as `famorassem` (family or assembly)—family for the `protected` C# keyword and assembly for the `internal` keyword. `famandassem` is also available with the IL code. Because of the demand for an AND combination, the C# team had some issues finding a good name for this, and finally it was decided to use `private protected` to restrict access from within the assembly to types that have an inheritance relationship but no types from any other assembly.

The following table lists all the access modifiers and their uses:

MODIFIER	APPLIES TO	DESCRIPTION
`public`	Any types or members	The item is visible to any other code.
`protected`	Any member of a type and any nested type	The item is visible only to the type and any derived type.
`internal`	Any types or members	The item is visible only within its containing assembly.
`private`	Any member of a type, and any nested type	The item is visible only inside the type to which it belongs.
`protected internal`	Any member of a type and any nested type	The item is visible to any code within its containing assembly and to any code inside a derived type.
`private protected`	Any members of a type and any nested type	The item is visible to the type and any derived type that is specified within the containing assembly.

Other Modifiers

The modifiers in the following table can be applied to members of types and have various uses. A few of these modifiers also make sense when applied to types:

MODIFIER	APPLIES TO	DESCRIPTION
`new`	Function members	The member hides an inherited member with the same signature.
`static`	All members	The member does not operate on a specific instance of the class. This is also known as *class member* instead of instance member.
`virtual`	Function members only	The member can be overridden by a derived class.

continues

(continued)

MODIFIER	APPLIES TO	DESCRIPTION
`abstract`	Function members only	A virtual member that defines the signature of the member but doesn't provide an implementation.
`override`	Function members only	The member overrides an inherited virtual or abstract member.
`sealed`	Classes, methods, and properties	For classes, the class cannot be inherited from. For properties and methods, the member overrides an inherited virtual member but cannot be overridden by any members in any derived classes. This must be used in conjunction with `override`.
`extern`	Static `[DllImport]` methods only	The member is implemented externally, in a different language. The use of this keyword is explained in Chapter 13, "Managed and Unmanaged Memory."

INHERITANCE WITH RECORDS

Chapter 3 discusses a new feature with C# 9: records. Behind the scenes, records are classes. However, you cannot derive a record from a class (other than the `object` type), and a class cannot derive from a record. However, records can derive from other records.

Let's change the shapes sample to use positional records. With the following code snippet, `Position` and `Size` are records that contain `X`, `Y`, `Width`, and `Height` properties with set init-only accessors as specified by the primary constructor. `Shape` is an abstract record with `Position` and `Size` properties, a `Draw` method, and a virtual `DisplayShape` method. As with classes, you can use modifiers with records, such as `abstract` and `virtual`. The previously specified `Clone` method is not needed with records because this is created automatically using the `record` keyword (code file `RecordsInheritance/Shape.cs`):

```
public record Position(int X, int Y);

public record Size(int Width, int Height);

public abstract record Shape(Position Position, Size Size)
{
  public void Draw() => DisplayShape();

  protected virtual void DisplayShape()
  {
    Console.WriteLine($"Shape with {Position} and {Size}");
  }
}
```

The `Rectangle` record derives from the `Shape` record. With the primary constructor syntax used with the `Rectangle` type, derivation from `Shape` passes the same values to the primary constructor of the `Shape`. Similar to the `Rectangle` class created earlier, in the `Rectangle` record, the `DisplayShape` method is overridden (code file `RecordsInheritance/ConcreteShapes.cs`):

```
public record Rectangle(Position Position, Size Size) : Shape(Position, Size)
{
  protected override void DisplayShape()
```

```
    {
      Console.WriteLine($"Rectangle at position {Position} with size {Size}");
    }
  }
```

With the top-level statements in the Program.cs file, a Rectangle and an Ellipse are created using primary constructors. The implementation of the Ellipse record is similar to the Rectangle record. The first rectangle created is cloned by using the built-in functionality, and with the new Rectangle, the Position property is set to a new value using the with expression. The with expression makes use of the init-only set accessors created from the primary constructor (code file RecordsInheritance/Program.cs):

```
Rectangle r1 = new(new Position(33, 22), new Size(200, 100));
Rectangle r2 = r1 with { Position = new Position(100, 22) };
Ellipse e1 = new(new Position(122, 200), new Size(40, 20));

DisplayShapes(r1, r2, e1);

void DisplayShapes(params Shape[] shapes)
{
  foreach (var shape in shapes)
  {
    shape.Draw();
  }
}
```

> **NOTE** *With future C# versions, the inheritance with records might be relaxed to allow inheritance from classes.*

USING INTERFACES

A class can derive from one class, and a record can derive from one record; you cannot use multiple inheritance with classes and records. You can use interfaces to bring multiple inheritance into C#. Both classes and records can implement multiple interfaces. Also, one interface can inherit from multiple interfaces.

Before C# 8, an interface never had any implementation. In the versions since C# 8, you can create an implementation with interfaces, but this is very different from the implementation with classes and records; interfaces cannot keep state, so fields or automatic properties are not possible. Because method implementation is only an additional feature of interfaces, let's keep this discussion for later in this chapter and first focus on the contract aspect of interfaces.

Predefined Interfaces

Let's take a look at some predefined interfaces and how they are used with .NET. Some C# keywords are even designed to work with particular predefined interfaces. The using statement and the using declaration (covered in detail in Chapter 13) use the IDisposable interface. This interface defines the method Dispose without any arguments and without return type. A class deriving from this interface needs to implement this Dispose method:

```
public IDisposable
{
  void Dispose();
}
```

The `using` statement uses this interface. You can use this statement with any class (here, the `Resource` class) implementing this interface:

```
using (Resource resource = new())
{
  // use the resource
}
```

The compiler converts the `using` statement to this code to invoke the `Dispose` method in the `finally` block of the `try`/`finally` statement:

```
Resource resource = new();
try
{
  // use the resource
}
finally
{
  resource.Dispose();
}
```

> **NOTE** *The* `try`/`finally` *block is covered in Chapter 10, "Errors and Exceptions."*

Another example where an interface is used with a language keyword is the `foreach` statement that's using the `IEnumerator` and `IEnumerable` interfaces. This code snippet

```
string[] names = { "James", "Jack", "Jochen" };
foreach (var name in names)
{
  Console.WriteLine(name);
}
```

is converted to access the `GetEnumerator` method of the `IEnumerable` interface and uses a `while` loop to access the `MoveNext` method and the `Current` property of the `IEnumerator` interface:

```
string[] names = { "James", "Jack", "Jochen" };
var enumerator = names.GetEnumerator();
while (enumerator.MoveNext())
{
  var name = enumerator.Current;
  Console.WriteLine(name);
}
```

> **NOTE** *Creating a custom implementation of the* `IEnumerable` *and* `IEnumerator` *interfaces with the help of the* `yield` *statement is covered in Chapter 6, "Arrays."*

Let's look at an example where an interface is used from a .NET class, and you can easily implement this interface. The interface `IComparable<T>` defines the `CompareTo` method to sort objects of the type you need to specify with the generic parameter `T`. This interface is used by various classes in .NET to order objects of any type:

```
public interface IComparable<in T>
```

```
{
  int CompareTo(T? other);
}
```

With the following code snippet, the record `Person` implements this interface specifying `Person` as a generic parameter. `Person` specifies the properties `FirstName` and `LastName`. The `CompareTo` method is defined to return 0 if both values (`this` and `other`) are the same, a value lower than 0 if `this` object should come before the `other` object, and a value greater than 0 if `other` should be first. Because the `string` type also implements `IComparable`, this implementation is used to compare the `LastName` properties. If the comparison on the last name returns `0`, a comparison is done on the `FirstName` property as well (code file `UsingInterfaces/Person.cs`):

```
public record Person(string FirstName, string LastName) : IComparable<Person>
{
  public int CompareTo(Person? other)
  {
    int compare = LastName.CompareTo(other?.LastName);
    if (compare is 0)
    {
      return FirstName.CompareTo(other?.FirstName);
    }
    return compare;
  }
}
```

With the top-level statements in `Program.cs`, three `Person` records are created within an array, and the array's `Sort` method is used to sort the elements in the array (code file `UsingInterfaces/Program.cs`):

```
Person p1 = new("Jackie", "Stewart");
Person p2 = new("Graham", "Hill");
Person p3 = new("Damon", "Hill");

Person[] people = { p1, p2, p3 };
Array.Sort(people);
foreach (var p in people)
{
  Console.WriteLine(p);
}
```

Running the application shows the `ToString` output of the record type in a sorted order:

```
Person { FirstName = Damon, LastName = Hill }
Person { FirstName = Graham, LastName = Hill }
Person { FirstName = Jackie, LastName = Stewart }
```

Interfaces can act as a contract. The record `Person` implements the `IComparable` contract that is used by the `Sort` method of the `Array` class. The `Array` class just needs to know the contract definition (the members of the interface) to know what it can use.

Dependency Injection with Interfaces

Let's create a custom interface. With the shapes sample, the `Shape` and `Rectangle` types used the `Console.WriteLine` method to write a message to the console:

```
protected virtual void DisplayShape()
{
  Console.WriteLine($"Shape with {Position} and {Size}");
}
```

This way, the method `DisplayShape` has a strong dependency on the `Console` class. To make this implementation independent of the `Console` class and to write to either the console or a file, you can define a contract such as the `ILogger` interface in the following code snippet. This interface specifies the `Log` method where a string can be passed as an argument (code file `UsingInterfaces/ILogger.cs`):

```
public interface ILogger
{
  void Log(string message);
}
```

A new version of the `Shape` class uses *constructor injection* where the interface is injected into an object of this class. In the constructor, the object passed with the parameter is assigned to the read-only property `Logger`. With the implementation of the `DisplayShape` method, the property of type `ILogger` is used to write a message (code file `UsingInterfaces/Shape.cs`):

```
public abstract class Shape
{
  public Shape(ILogger logger)
  {
    Logger = logger;
  }

  protected ILogger Logger { get; }
  public Position? Position { get; init; }
  public Size? Size { get; init; }

  public void Draw() => DisplayShape();

  protected virtual void DisplayShape()
  {
    Logger.Log($"Shape with {Position} and {Size}");
  }
}
```

With a concrete implementation of the abstract `Shape` class, in the constructor, the `ILogger` interface is forwarded to the constructor of the base class. With the `DisplayShape` method, the protected property `Logger` is used from the base class (code file `UsingInterfaces/ConcreteShapes.cs`):

```
public class Ellipse : Shape
{
  public Ellipse(ILogger logger) : base(logger) { }

  protected override void DisplayShape()
  {
    Logger.Log($"Ellipse at position {Position} with size {Size}");
  }
}
```

Next, a concrete implementation of the `ILogger` interface is required. One way you can implement writing a message to the console is with the `ConsoleLogger` class. This class implements the `ILogger` interface to write a message to the console (code file `UsingInterfaces/ConsoleLogger.cs`):

```
public class ConsoleLogger : ILogger
{
  public void Log(string message) => Console.WriteLine(message);
}
```

> **NOTE** *Using the* `ILogger` *interface from the* `Microsoft.Extensions.Logging` *namespace is discussed in Chapter 16, "Diagnostics and Metrics."*

For creating a `Rectangle`, the `ConsoleLogger` can be created on passing an instance to implement the `ILogger` interface (code file `UsingInterfaces/Program.cs`):

```
Ellipse e1 = new(new ConsoleLogger())
{
  Position = new(20, 30),
  Size = new(100, 120)
};
r1.Draw();
```

> **NOTE** *With dependency injection, the responsibility is turned over. Instead of having a strong dependency with the implementation of the shape for the* `Console` *class, the responsibility for what is used is turned over outside of the* `Shape` *type. This way what is used can be specified from the outside. This is also known as the Hollywood Principle—"Don't call us, we call you." Dependency injection makes unit testing easier because dependencies can be easily replaced with mock types. Another advantage when using dependency injection is that you can create platform-specific implementations. For example, showing a message box is different with the Universal Windows Platform (*`MessageDialog.ShowAsync`*), WPF (*`MessageBox.Show`*), and Xamarin.Forms (*`Page.Alert`*). With a common view model, you can use the interface* `IDialogService` *and define different implementations with the different platforms. Read more about dependency injection using a dependency injection container in Chapter 15, "Dependency Injection and Configuration." Unit testing is covered in Chapter 23, "Tests."*

Explicit and Implicit Implemented Interfaces

Interfaces can be explicitly or implicitly implemented. With the example so far, you've seen implicitly implemented interfaces, such as with the `ConsoleLogger` class:

```
public class ConsoleLogger : ILogger
{
  public void Log(string message) => Console.WriteLine(message);
}
```

With an explicit interface implementation, the member implemented doesn't have an access modifier and has the interface prefixed to the method name:

```
public class ConsoleLogger : ILogger
{
  void ILogger.Log(string message) => Console.WriteLine(message);
}
```

With an explicit interface implementation, the interface is not accessible when you use a variable of type `ConsoleLogger` (it's not public). If you use a variable of the interface type (`ILogger`), you can invoke the `Log` method; the contract of the interface is fulfilled. You can also cast the `ConsoleLogger` variable to the interface `ILogger` to invoke this method.

Why would you want to do this? One reason is to resolve a conflict. If different interfaces define the same method signature, your class needs to implement all these interfaces, and the implementations need to differ, you can use explicit interface implementation.

Another reason to use explicit interface implementation is to hide the interface method from code outside of the class but still fulfill the contract from the interface. An example is the `StringCollection` class from the `System.Collections.Specialized` namespace and the `IList` interface. One of the members that's defined by the `IList` interface is the `Add` method:

```
int Add(object? value);
```

The `StringCollection` class is optimized for strings and thus prefers to use the string type with the `Add` method:

```
public int Add(string? value);
```

The version to pass an object is hidden from the `StringCollection` class because the `StringCollection` class has an explicit interface implementation with this method. To use this type directly, you just pass a string parameter. If a method uses `IList` as a parameter, then you can use any object that implements `IList` for that parameter. In particular, you can use a `StringCollection` for the parameter because that class still implements that interface.

Comparing Interfaces and Classes

Now that you've seen the foundations of interfaces, let's compare interfaces, classes, records, and structs with regard to object orientation:

- You can declare a variable of the type of all these C# constructs. You can declare a variable of a class, an interface, a record, or a struct.
- You can instantiate a new object with classes, records, and structs. You cannot instantiate a new object with an abstract class or an interface.
- With a class, you can derive from a base class. With a record, you can derive from a base record. Both with classes and records, implementation inheritance is supported. Structs don't support inheritance.
- Classes, records, and structs can implement multiple interfaces. Implementing interfaces is not possible with ref structs.

Default Interface Methods

Before C# 8, changing an interface was always a breaking change. Even just adding a member to an interface is a breaking change. The type implementing this interface needs to implement this new interface member. Because of this, many .NET libraries are built with abstract base classes. When you add a new member to an abstract base class, if it's not an abstract member, it is not a breaking change. With Microsoft's Component Object Model (COM), which is based on interfaces, always a new interface was defined when a breaking change was introduced—for example, `IViewObject`, `IViewObjectEx`, `IViewObject2`, `IViewObject3`.

As of C# 8, interfaces can have implementations. However, you need to be aware where you can use this feature. C# 8 is supported by .NET Core 3.x. With older technologies, you can change the compiler version at your own risk. To support default interface members, a runtime change is required. This runtime change is available only with .NET Core 3.x+ and .NET Standard 2.1+. You cannot use default interface members with .NET Framework applications or UWP applications without .NET 5 support.

Avoiding Breaking Changes

Let's get into the main feature of default interface members to avoid breaking changes. In a previous code sample, the `ILogger` interface has been specified:

```
public interface ILogger
{
```

```
  void Log(string message);
}
```

If you add any member without implementation, the `ConsoleLogger` class needs to be updated. To avoid a breaking change, an implementation to the new `Log` method with the `Exception` parameter is added. With the implementation, the previous `Log` method is invoked by passing a `string` (code file `DefaultInterfaceMethods/ILogger.cs`):

```
public interface ILogger
{
  void Log(string message);
  public void Log(Exception ex) => Log(ex.Message);
}
```

> **NOTE** *The implementation of the* `Log` *method has the* `public` *access modifier applied. With interface members,* `public` *is the default, so this access modifier is not required. However, with implementations in the interface, you can use the same modifiers you've seen with classes, including* `virtual`, `abstract`, `sealed`, *and so on.*

The application can be built without changing the implementation of the `ConsoleLogger` class. If a variable of the interface type is used, both `Log` methods can be invoked: the `Log` method with the string parameter and the `Log` method with the `Exception` parameter (code file `DefaultInterfaceMethods/Program.cs`):

```
ILogger logger = new ConsoleLogger();
logger.Log("message");
logger.Log(new Exception("sample exception"));
```

With a new implementation of the `ConsoleLogger` class, a different implementation of the new `Log` method defined with the `ILogger` interface can be created. In this case, using the `ILogger` interface invokes the method implemented with the `ConsoleLogger` class. The method is implemented with explicit interface implementation but could be implemented with implicit interface implementation as well (code file `DefaultInterfaceMethods/ConsoleLogger.cs`):

```
public class ConsoleLogger : ILogger
{
  public void Log(string message) => Console.WriteLine(message);

  void ILogger.Log(Exception ex)
  {
    Console.WriteLine(
      $"exception type: {ex.GetType().Name}, message: {ex.Message}");
  }
}
```

Traits with C#

Default interface members can be used to implement traits with C#. *Traits* allow you to define methods for a group of types. One way to implement traits is with extension methods; the other option is using default interface methods.

With Language Integrated Query (LINQ), many LINQ operators have been implemented with extension methods. With this new feature, it would be possible to implement these methods with default interface members instead.

NOTE *Extension methods are introduced in Chapter 3. Chapter 9, "Language Integrated Query," covers all the extension methods implemented with LINQ.*

To demonstrate this, the `IEnumerableEx<T>` interface is defined that derives from the interface `IEnumerable<T>`. Deriving from this interface, `IEnumerableEx<T>` specifies the same contract as the base interface, but the `Where` method is added. This method receives a delegate parameter to pass a predicate method that returns a Boolean value, iterates through all the items, and invokes the method referenced by the predicate. If the predicate returns true, the Where method returns the item with `yield return`.

```
using System;
using System.Collections.Generic;

public interface IEnumerableEx<T> : IEnumerable<T>
{
  public IEnumerable<T> Where(Func<T, bool> pred)
  {
    foreach (T item in this)
    {
      if (pred(item))
      {
        yield return item;
      }
    }
  }
}
```

NOTE *The* `yield` *statement is covered in detail in Chapter 6.*

Now you need a collection to implement the interface `IEnumerableEx<T>`. You can do this easily by creating a new collection type, `MyCollection`, that derives from the `Collection<T>` base class defined in the `System.Collections.ObjectModel` namespace. Because the `Collection<T>` class already implements the interface `IEnumerable<T>`, no additional implementation is needed to support `IEnumerableEx<T>` (code file `DefaultInterfaceMethods/MyCollection.cs`):

```
class MyCollection<T> : Collection<T>, IEnumerableEx<T>
{
}
```

With this in place, a collection of type `MyCollections<string>` is created that's filled with names. A lambda expression that returns a Boolean value and receives a string is passed to the `Where` method that's defined with the interface. The `foreach` statement iterates through the result and only displays the names starting with `J` (code file `DefaultInterfaceMethods/Program.cs`):

```
IEnumerableEx<string> names = new MyCollection<string>
  { "James", "Jack", "Jochen", "Sebastian", "Lewis", "Juan" };

var jNames = names.Where(n => n.StartsWith("J"));
foreach (var name in jNames)
{
  Console.WriteLine(name);
}
```

> **NOTE** *When you invoke default interface members, you always need a variable of the interface type, similar to explicitly implemented interfaces.*
>
> *What cannot be done with interfaces and default interface members is to add members that keep state. Fields, events (with delegates), and auto properties add state—these members are not allowed. If state is required, you should use abstract classes instead.*

GENERICS

One way to reduce the code you need to write is by using inheritance and adding functionality to base classes. Another way is to create generics where a type parameter is used, which allows specifying the type when instantiating the generic (which can also be combined with inheritance).

Let's get into an example to create a linked list of objects where every item references the next and previous items. The first generic type created is a record. The generic type parameter is specified using angle brackets. `T` is the placeholder type parameter name. With the primary constructor, a property with an init-only set accessor is created. The record has two additional properties, `Next` and `Prev`, to reference the next and previous items. With these additional properties, the `internal` access modifier is used to allow calling the set accessor only from within the same assembly (code file `GenericTypes/LinkedListNode.cs`):

```
public record LinkedListNode<T>(T Value)
{
  public LinkedListNode<T>? Next { get; internal set; }
  public LinkedListNode<T>? Prev { get; internal set; }
  public override string? ToString() => Value?.ToString();
}
```

> **NOTE** *Because the* `LinkedListNode` *type is a record, it's important to override the* `ToString` *method. With the default implementation of the* `ToString` *method, the value of all property members is shown, which invokes* `ToString` *with every property value. Because the* `Next` *and* `Prev` *properties reference other objects, a stack overflow can occur.*

The generic class `LinkedList` contains the properties `First` and `Last` to access the first and last elements of the list, the method `AddLast` to add a new node at the end of the list, and an implementation of the `IEnumerable<T>` interface, which allows iterating through all elements (code file `GenericTypes/LinkedList.cs`):

```
public class LinkedList<T> : IEnumerable<T>
{
  public LinkedListNode<T>? First { get; private set; }
  public LinkedListNode<T>? Last { get; private set; }
  public LinkedListNode<T> AddLast(T node)
  {
    LinkedListNode<T> newNode = new(node);
    if (First is null || Last is null)
    {
      First = newNode;
```

```
      Last = First;
    }
    else
    {
      newNode.Prev = Last;
      LinkedListNode<T> previous = Last;
      Last.Next = newNode;
      Last = newNode;
    }
    return newNode;
  }

  public IEnumerator<T> GetEnumerator()
  {
    LinkedListNode<T>? current = First;
    while (current is not null)
    {
      yield return current.Value;
      current = current.Next;
    }
  }

  IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
}
```

In the generated `Main` method, the `LinkedList` is initiated by using the `int` type by using the `string` type, a tuple, and a record. `LinkedList` works with any type (code file `GenericTypes/Program.cs`):

```
LinkedList<int> list1 = new();
list1.AddLast(1);
list1.AddLast(3);
list1.AddLast(2);

foreach (var item in list1)
{
  Console.WriteLine(item);
}
Console.WriteLine();

LinkedList<string> list2 = new();
list2.AddLast("two");
list2.AddLast("four");
list2.AddLast("six");

Console.WriteLine(list2.Last);

LinkedList<(int, int)> list3 = new();
list3.AddLast((1, 2));
list3.AddLast((3, 4));
foreach (var item in list3)
{
  Console.WriteLine(item);
}
Console.WriteLine();
```

```
LinkedList<Person> list4 = new();
list4.AddLast(new Person("Stephanie", "Nagel"));
list4.AddLast(new Person("Matthias", "Nagel"));
list4.AddLast(new Person("Katharina", "Nagel"));

// show the first
Console.WriteLine(list4.First);

public record Person(string FirstName, string LastName);
```

Constraints

With the previous implementation of the `LinkedListNode<T>` and `LinkedList<T>` types there was not a special requirement on the generic type; any type can be used. This prevents you from using any nonobject members with the implementation. The compiler doesn't accept invoking any property or method on the generic type `T`.

Adding the `DisplayAllTitles` method to the `LinkedList<T>` class results in a compiler error. `T` does not contain a definition for `Title`, and no accessible extension method `Title` accepting a first argument of type `T` could be found (code file `GenericTypesWithConstraints/LinkedList.cs`):

```
public void DisplayAllTitles()
{
  foreach (T item in this)
  {
    Console.WriteLine(item.Title);
  }
}
```

To resolve this, the interface `ITitle` is specified that defines a `Title` property that needs to be implemented with the implementation of this interface:

```
public interface ITitle
{
  string Title { get; }
}
```

Defining the generic `LinkedList<T>`, now the constraint for the generic type `T`, can be specified to implement the interface `ITitle`. Constraints are specified with the `where` keyword followed by the requirement on the type:

```
public class LinkedList<T> : IEnumerable<T>
    where T : ITitle
{
  //...
}
```

With this change in place, the `DisplayAllTitles` method compiles. This method uses the members specified by the `ITitle` interface, and this is a requirement on the generic type. You can no longer use `int` and `string` for the generic type parameter, but the `Person` record can be changed to implement this constraint (code file `GenericTypesWithConstraints/Program.cs`):

```
public record Person(string FirstName, string LastName, string Title)
  : ITitle { }
```

The following table lists the constraints you can specify with a generic:

CONSTRAINT	DESCRIPTION
`where T : struct`	With a struct constraint, `T` must be a value type.
`where T : class`	With a class constraint, `T` must be a reference type.
`where T : class?`	`T` must be a nullable or a non-nullable reference type.
`where T : notnull`	`T` must be a non-nullable type. This can be a value or a reference type.
`where T : unmanaged`	`T` must be a non-nullable unmanaged type.
`where T : IFoo`	This specifies that the type `T` is required to implement interface `IFoo`.
`where T : Foo`	This specifies that the type `T` is required to derive from base class `Foo`.
`where T : new()`	A constructor constraint; this specifies that `T` must have a parameterless constructor. You cannot specify a constraint for constructors with parameters.
`where T1 : T2`	With constraints, it is also possible to specify that type `T1` derives from a generic type `T2`.

SUMMARY

This chapter described how to code inheritance in C#. You saw the rich support for both implementing multiple interfaces and single inheritance with classes and records. You saw how C# provides a number of useful syntactical constructs designed to assist in making code more robust, which includes different access modifiers, and the concept of nonvirtual and virtual methods. You also saw the new feature for interfaces, which allows adding code implementation. Generics have been covered as another concept to reuse code.

The next chapter continues with all the C# operators and casts.